Implementing Functions in SMoL

Environments and Closures

Functions as First-Class Values

- Functions don't *need* names in SMoL (Modern PLs)

  so instead of

```
fun f(x):
  x + x
```

The language supports anonymous functions (lambdas), like:

```
sq = lam(y): y + y
```

Functions as First-Class Values

```
sq = lam(y): y + y
```

- Create an anonymous function (lam)
- Bind it to a name (sq = ...)
- By separating them, the language stays simpler; Naming and function-creation are independent ideas.

Two New Constructs in the Abstract Syntax Tree (AST)

sq = lam(y): y + y

Construct	Purpose	Example
`lamE`	**Introduce** (create) a function	`lam(x): x * x`
`appE`	**Eliminate** (apply/call) a function	`sq(3)`

S-expression syntax

```
{let1 {f {lam x {+ x
x}}}
    {f 3}}
```

AST

```
(let1E 'f (lamE 'x (plusE (varE 'x) (varE 'x)))
 (appE (varE 'f) (numE 3)))
```

Evaluation value

- lam(x): x + x -> is an object function, it is represented by a new kind of value (type), not a number.

lambda evaluates to a **procedure/function object.**

Thus, the interpreter needs a new funV value type

funV (var : Symbol) (body : Exp) and something else

Evaluation value (Example)

```
x = 1
f = lam(y): x
x = 2
f(10)
```

Output= ?, The correct value should be the one following the substitution. But, substitution is an old stuff.

Example, let’s use substitution

x = 1
f = lam(y): x
x = 2
f(10)
---
f = lam(y): 1
x = 2
f(10)
---
x = 2
(lam(y): 1)(10)

(lam(y): 1)(10)

1

Environments and closures

Closure is a function value that **remembers the environment at the moment it was created**. The funV type gets extended to store the environment:

funV (var : Symbol) (body : Exp) (nv : Env)

Creating a lambda now captures the current environment:

[(lamE v b) → (funV v b nv)] ← nv is saved here!

Example (SMoL behavior)

```
x = 3
f = lambda y: x + y
x = 2
print(f(4))
```

```
env = {}
```

```
env = { x ⟶ 3 }
```

```
After f = lambda y: x + y:
```

```
env = { x=3,  f=closure }
                  ↓
            closure = {
              param: y
              body:  x + y
              nv:    { x=3 }   ← snapshot frozen here!
            }
```

```
After x = 3:
    x ⟶ 3

After f = lambda y: x + y:
    x ⟶ 3
    f ⟶ lambda (has a pointer to x, not to 3)

After x = 2:
    x ⟶ 2        ← x now points somewhere else
    f ⟶ lambda (still pointing to x, which is now 2)

f(4):
    looks up x → 2
    2 + 4 = 6
```

```
env = { x=2,  f=closure }
                  ↓
            closure = {
              param: y
              body:  x + y
              nv:    { x=3 }   ← completely unchanged!
            }
```

```
y = 4

look up x in closure's OWN nv:
  { x=3 } → x = 3            ← ignores the outer x=2

evaluate x + y:
  3 + 4 = 7
```

Example 2

```
(run '{let1 {x 100}
            {let1 {f {lam y {+ x y}}}
                  {let1 {x 999}
                        {f 1}}}})
```

- Step 1, start, env = {}
- Step 2 — evaluate {let1 {x 100} ...}, env = { x=100
- Step 3 — evaluate {let1 {f {lam y {+ x y}}} ...}
  closure = (funV 'y (+ x y) { x=100 }) nv frozen inside!

Then bind f to that closure, extend the environment:
env = { f=funV, x=100 }

- Step 4 — evaluate {let1 {x 999} ...} Bind x = 999, extend the environment:
  env = { x=999, f=funV, x=100 }
  ↑ new ↑ old x=100 still there but shadowed by x=999
- Step 5, Evaluate {f 1}, Now the two environments meet:
  outer nv (caller's) = { x=999, f=funV, x=100 } ← thrown away!
  closure's nv* = { x=100 } ← used instead! (STATIC SCOPE)

extend nv* 'y (numV 1)
→ { y=1, x=100 }

The new language supports

- numbers
- variables
- addition
- let bindings
- anonymous functions (lam)
- function application (app)

The syntax: s-expressions

```
3                              ; a number
x                              ; a variable
{+ 3 4}                        ; addition
{lam x {+ x x}}                ; a function
{f 3}                          ; function call
{let1 {x 3} {+ x x}}          ; let binding
```

Grammar (conceptual)

```
<expr> ::= <num>                                numeric literal
         | <var>                                variable reference
         | { + <expr> <expr> }                  addition
         | { lam <var> <expr> }                 anonymous function
         | { <expr> <expr> }                    function application
         | { let1 { <var> <expr> } <expr> }     local binding
```

`<num>` — any number literal: `3`, `-7`, `42`

`<var>` — any symbol used as a name: `x`, `f`, `sq`

`{ + <expr> <expr> }` — add two expressions: `{+ x 1}`, `{+ {+ 1 2} 3}`

`{ lam <var> <expr> }` — create a function: `{lam x {+ x x}}`

`{ <expr> <expr> }` — call a function: `{f 3}`, `{{lam x x} 5}`

`{ let1 { <var> <expr> } <expr> }` — bind a name locally: `{let1 {x 3} {+ x x}}`

AST

```
(define-type Exp
  [numE  (n    : Number)]          ; a number literal
  [varE  (name : Symbol)]          ; a variable reference
  [plusE (left : Exp)              ; addition
         (right : Exp)]
  [lamE  (var  : Symbol)           ; anonymous function
         (body : Exp)]
  [appE  (fun  : Exp)              ; function application
         (arg  : Exp)]
  [let1E (var  : Symbol)           ; local binding
         (val  : Exp)
         (body : Exp)])
```

```
<expr> ::= <num>                                  numeric literal
         | <var>                                  variable reference
         | { + <expr> <expr> }                    addition
         | { lam <var> <expr> }                   anonymous function
         | { <expr> <expr> }                      function application
         | { let1 { <var> <expr> } <expr> }       local binding
```

Type

```
(define-type Value
  [numV (n    : Number)]
  [funV (var  : Symbol)
        (body : Exp)
        (nv   : Env)])      ← closure captures env at runtime
```

Environment – a list in this case

```
(define-type Binding
  [bind (name : Symbol) (val : Value)])

(define-type-alias Env (Listof Binding))

(define empty-env : Env empty)
```

Helpers

```
(define (extend [env : Env] [name : Symbol] [val : Value]) : Env
  (cons (bind name val) env))

(define (lookup [name : Symbol] [env : Env]) : Value
  (cond
    [(empty? env)                                (error name "unbound variable")]
    [(symbol=? name (bind-name (first env))) (bind-val (first env))]
    [else                                        (lookup name (rest env))]))

(define (add [l : Value] [r : Value]) : Value
  (cond
    [(and (numV? l) (numV? r)) (numV (+ (numV-n l) (numV-n r)))]
    [else (error 'add "not a number")]))
```

```
l       = [ let1, {sq {lam x {+ x x}}}, {sq 5} ]
            first       second                third
```

Parse the binding `{sq {lam x {+ x x}}}`:

```
binding  = [ sq,  {lam x {+ x x}} ]
var-name = 'sq
var-val  = parse({lam x {+ x x}})
```

Parse `{lam x {+ x x}}`:

```
head = 'lam  →  lamE branch
  var  = 'x
  body = parse({+ x x})
           head = '+  →  plusE branch
             left  = parse(x) = (varE 'x)
             right = parse(x) = (varE 'x)
           = (plusE (varE 'x) (varE 'x))

→ (lamE 'x (plusE (varE 'x) (varE 'x)))
```

Parse the body `{sq 5}`:

```
head = 'sq  →  else branch  →  appE
  fun = parse(sq) = (varE 'sq)
  arg = parse(5)  = (numE 5)
→ (appE (varE 'sq) (numE 5))
```

Assemble the full let1E:

```
(let1E 'sq
       (lamE 'x (plusE (varE 'x) (varE 'x)))
       (appE (varE 'sq) (numE 5)))
```

PARSER

{let1 {sq {lam x {+ x x}}} {sq 5}}

```
(cond
  [(s-exp-number? s) (numE (s-exp->number s))]
  [(s-exp-symbol? s) (varE (s-exp->symbol s))]
  [(s-exp-list? s)
   (let ([l (s-exp->list s)])
     (cond
       [(symbol=? 'lam  (s-exp->symbol (first l)))
        (lamE (s-exp->symbol (second l))
              (parse (third l)))]
       [(symbol=? '+    (s-exp->symbol (first l)))
        (plusE (parse (second l))
               (parse (third l)))]
       [(symbol=? 'let1 (s-exp->symbol (first l)))
        (let ([b (s-exp->list (second l))])
          (let1E (s-exp->symbol (first b))
                 (parse (second b))
                 (parse (third l))))]
       [else
        (appE (parse (first l))
              (parse (second l)))]))]
  [else (error 'parse "not a number, symbol, or list")]))
```

Interpreter

```
(define (run [s : S-Exp]) : Value
  (interp (parse s) empty-env))
```

```
(varE
'x)
```

```
run('x)
  └─ interp((varE 'x), {})
       └─ lookup('x, {})
            └─ error: "unbound variable"   ×
```

```
(let1E 'sq
       (lamE 'x (plusE (varE 'x) (varE 'x)))
       (appE (varE 'sq) (numE 5)))
```

```
(define (interp [e : Exp] [nv : Env]) : Value
  (type-case Exp e
    [(numE n)    (numV n)]
    [(varE s)    (lookup s nv)]
    [(plusE l r) (add (interp l nv) (interp r nv))]
    [(lamE v b)  (funV v b nv)]          ← env frozen here
    [(appE f a)
     (let ([fv (interp f nv)]
           [av (interp a nv)])
       (type-case Value fv
         [(funV v b nv*)                 ← nv* shadows outer nv
          (interp b (extend nv* v av))]
         [else (error 'app "not a function")]))]
    [(let1E var val body)
     (interp body (extend nv var (interp val nv)))]))
```

Interpreter

```
(define (run [s : S-Exp]) : Value
  (interp (parse s) empty-env))
```

Run the pipeline: `x

```
(let1E 'sq
       (lamE 'x (plusE (varE 'x) (varE 'x)))
       (appE (varE 'sq) (numE 5)))
```

```
(define (interp [e : Exp] [nv : Env]) : Value
  (type-case Exp e
    [(numE n)    (numV n)]
    [(varE s)    (lookup s nv)]
    [(plusE l r) (add (interp l nv) (interp r nv))]
    [(lamE v b)  (funV v b nv)]          ← env frozen here
    [(appE f a)
     (let ([fv (interp f nv)]
           [av (interp a nv)])
       (type-case Value fv
         [(funV v b nv*)                 ← nv* shadows outer nv
          (interp b (extend nv* v av))]
         [else (error 'app "not a function")]))]
    [(let1E var val body)
     (interp body (extend nv var (interp val nv)))]))
```